Path Planning Algorithms Analysis

Yuhao Liu, Yahsiu Hsieh
Department of Electrical & Computer Engineering
University of California, San Diego

***Abstract*—This paper will focus on comparing the performance of search-based and sampling-based planning algorithms in a 3-D Euclidean space. Four algorithms, D* Lite, RRT, RRT*, and RRTConnect, will be discussed.**

I. Introduction

Motion Planning is a term used in robotics is to find a sequence of valid configurations that moves the robot from the starting point to destination. It has several robotics applications, such as autonomy, automation, and robot design in CAD software.

This project will focus on comparing the performance of search-based and sampling-based planning algorithms in 3-D Euclidean space. For a given map, we use each algorithm to get a feasible path from a start point to a goal point. We then demonstrate the results and discuss the difference according to the processing time, parameter settings, and the path length. The environment is a 3-D environments described by a rectangular outer boundary and a set of rectangular obstacle blocks.

We introduce four algorithms,which are D* Lite, RRT, RRT*, and RRT Connect. D* Lite algorithm is one of Incremental Search algorithms, which reuses the data from the previous search to reduce the overall computation for planning. D* Lite is designed to solve the planning problem in a dynamic, unknown map. On the other hand, Rapidly Exploring Random Tree, or RRT, is one of the most common Sample-based algorithm. RRT constructs a tree from random samples with root. The tree is grown until it contains a path to the goal. Since it generates a path according to the start position and the goal position, it is well-suited for single-shot planning. In this project, we would delve into RRT and also explore its extension RRT* and RRT Connect.

The rest of paper is as follows. First we give the detailed formulations of Motion Planning problem in Section II. Technical approaches are introduced in Section III. And at last the experiment and results are presented in Section IV.

II. Problem Formulation

In this section, we formulate this problem as a deterministic shortest path problem and consider following basic elements.

Let $\mathbb{X} = (0,1)^d$ be the configuration space, where $d \in \mathbb{N}, d \geqslant 2$. Let $\mathbb{X}_{obs}$ be the obstacle region, such that $\mathbb{X} \backslash \mathbb{X}_{obs}$ is an open set, and denote the obstacle-free space as $\mathbb{X}_{free} = cl(\mathbb{X} \backslash \mathbb{X}_{obs})$, where $cl(\cdot)$ denotes the closure of a set. The initial condition x_{init} is an element of $\mathbb{X}_{free}$, and the goal region $\mathbb{X}_{goal}$ is an open subset of $\mathbb{X}_{free}$. A path planning problem is defined by a triplet $(\mathbb{X}_{free}, x_{init}, \mathbb{X}_{goal})$.

A function $\sigma : [0,1] \to \mathbb{R}^d$ of bounded variation is called a

- Path, if it is continuous;
- Collision-free path, if it is a path, and $\sigma(\tau) \in \mathbb{X}_{free}$, $\forall \tau \in [0,1]$;
- Feasible path if it is a collision-free path, $\sigma(0) = x_{init}$, and $\sigma(1) = cl(\mathbb{X}_{goal})$.

A feasible path planning problem is that, given a path planning problem $(\mathbb{X}_{free}, x_{init}, \mathbb{X}_{goal})$, find a feasible path $\sigma : [0,1] \to \mathbb{X}_{free}$ such that $\sigma(0) = x_{init}$ and $\sigma(1) \in cl(\mathbb{X}_{goal})$, if one exists. If no such path exists, report failure.

III. Technical Approach

In this section we will first discuss about how a search-based planning algorithm, D* Lite, is designed in this project, and then describe the sampling-based planning algorithms.

A. D* Lite Algorithm

D* Lite is an efficient repeated best-first search algorithm that traverses a graph with changing edge weights.[1] The name of D* comes from the algorithm A*, and the "D" stands for dynamic in the sense that edge cost changes. It repeatedly determines shortest paths between the current vertex of the robot and the goal vertex as the edge costs of a graph change while the robot moves towards the goal vertex. [2]

Some characteristics of D* Lite includes: D* Lite reuses the information as possible from the last computation instead of re-planning the path from scratch. Also, D* lite makes no assumption about how the edge costs changes. It can be used to solve the goal-directed navigation problem in unknown terrain. Since the terrain in this project is 3-dimensional, it is usually modeled as an 26-connected graph; however, we simplify the graph to 4-connected one, which contains only 6 directions, such as up, down left, right, forward and backward. In the following sections, we would discuss several core ideas of the algorithm

B. D* Lite: Reformulation

In previous projects, we have learned A* algorithm. The g-value cost, which is defined as the optimal path cost from the start point to the current node. Nevertheless, since that we would explore the optimal path while moving the agent, which would change the agent position, the path cost (g-value) is not preserved.

To preserve the path cost of a given node in this scenario, D* Lite changes the definition of g-value, which calculates the path cost from the current node to the “goal” point. As a result, the path cost g-value is preserved, and we can reuse this value for the next iteration even though the start node has changed.

C. D* Lite: Heap Recording

Whenever the agent moves and encounters a change in edge costs, the $main()$ needs to recalculate the priorities of the nodes in the priority queue since that the g value no longer holds due to the fact that it was computed with respect to the old position of the agent. Note that it is expensive to record repeated priority queue with large number of nodes. Therefore, D* Lite take advantage of the solution from the D* to avoid having to reorder the priority queue. The heuristic $h(n, n')$ needs to be non-negative, admissible, and consistent.

After the robot moves from node to some node , we know that the first component in the key value of a node, which is $min(g(n), rhs(n)) + h(n_{start}, n) + k_m$, would have decreased by at most $h(n, n')$. To keep the order in priority queue correct, the $h(n, n')$ term should add back to the first component. This term is controlled by the key modifier k_m.

D. D* Lite: Dynamic Simulation

Since that D* Lite is suitable for dynamic-map path planning, in our implementation, we simulate the region of observations of a agent as followed: First, we build the whole map using the information we have and set this map as the "global map." We then create a blank map (called "local map") from the perspective of the agent. In the beginning, the agent could only observe a cube-like region around it. The size is customized. When it starts to move, it would record the traversed region in the local map. The D* Lite would use the local map to plan a feasible map. We assume that there is no noise on the observation. The global map and the local map are boolean arrays, where false grid refers to the free space while true one indicates a obstacle.

Algorithm 1 illustrates the general idea of D* Lite algorithm for this particular problem.

Algorithm 1 D* Lite Algorithm

```
1: $g(n) \leftarrow \infty$, $rhs(n \neq n_{goal}) \leftarrow \infty$, $Queue \leftarrow []$
2: $rhs(n_{goal}) \leftarrow 0$, $k_m \leftarrow 0$, $n_{goal} = n_{start}$
3: while $n_{goal}$ is not reached do
4:     if $topKey > Key(n_{start})$ and $g(n_{last}) = rhs(n_{last})$ then
5:         Generate the optimal path
6:         Follow the path until the map is updated
7:         $k_m \leftarrow k_m + h(n_{last}, n_{start})$
8:         Update $rhs(obstacle)$
9:         $updateNeighbor(obstacle)$
10:    end if
11:    $currNode \leftarrow Queue.pop()$
12:    $updateG(currNode)$
13:    $updateNeighbor(currNode)$
14: end while
```

E. RRT Algorithm

In this section, we will introduce RRT and discuss the parameters we specified for the algorithm.

Algorithm 2 Key

```
1: $cost \leftarrow min(g(n), rhs(n))$
2: return [$cost + h(n_{start}, n) + k_m$; $cost$]
```

Algorithm 3 updateG

```
1: if $g(n) > rhs(n)$ then
2:     $g(n) = rhs(n)$
3: else if $g(n) < rhs(n)$ then
4:     $g(n) = \infty$
5: end if
```

The premise of RRT is actually quite straight forward. Points are randomly generated and connected to the closest available node. Each time a vertex is created, a check must be made that the vertex lies outside of an obstacle. Furthermore, chaining the vertex to its closest neighbor must also avoid obstacles. The algorithm ends when a node is generated within the goal region, or a limit is hit.

We set our length of the search tree edge as 0.5, and also set the goal bias parameter to 0.05, which means the fraction of time the goal is picked as the state to expand towards. After getting a sample, we will check if that sample is able to be connected from the tree, and we will check if the path is in collision with resolution 0.1 to guarantee the resulting solution path is collision free.

Although the RRT algorithm ensures probabilistic completeness, it cannot guarantee finding the most optimal path. Therefore, in section IV, we can see that the resulting path is pretty ugly. In the next section, we will introduce RRT* algorithm, which guarantee finding the optimal path.

Algorithm 5 illustrates the general idea of RRT algorithm for this particular problem, where V is the vertex and E is the tree edge.

F. RRT* Algorithm

In this section, we will introduce RRT* and discuss the parameters we specified for the algorithm.

RRT* is an optimized version of RRT. When the number of nodes approaches infinity, the RRT* algorithm will deliver the shortest possible path to the goal. The basic principle of RRT* is the same as RRT, but two key additions to the algorithm result in significantly different results.

First, RRT* records the distance each vertex has traveled relative to its parent vertex. This is referred to as the $Cost()$ of the vertex. After the closest node is found in the graph, a neighborhood of vertices in a fixed radius from the new node are examined. If a node with a cheaper $Cost()$ than the proximal node is found, the cheaper node replaces the proximal node.

The second difference RRT* adds is the rewiring of the tree. After a vertex has been connected to the cheapest neighbor, the neighbors are again examined. Neighbors are checked if being rewired to the newly added vertex will make their cost decrease. If the cost does indeed decrease, the neighbor is

Algorithm 4 updateNeighbor

```
1: for $n' \in Children(n)$ do
2:     $rhs(n) \leftarrow min_{n' \in Children(n)}(c(n, n') + g(n'))$
3:     if $g(n) \neq rhs(n)$ then
4:         $Queue.insert(n)$
5:     end if
6: end for
```

Algorithm 5 RRT Algorithm

```
1: $V \leftarrow \{x_s\}, E \leftarrow \emptyset$
2: for $i = 1 \ldots n$ do
3:     $x_{rand} \leftarrow SampleFree()$
4:     $x_{nearest} \leftarrow Nearest((V, E), x_{rand})$
5:     $x_{new} \leftarrow Steer(x_{nearest}, x_{rand})$
6:     if $CollisionFree(x_{nearest}, x_{new})$ then
7:         $V \leftarrow V \cup \{x_{new}\}$
8:         $E \leftarrow E \cup \{x_{(nearest}, x_{new})\}$
9:     end if
10: end for
11: return $G = (V, E)$
```

rewired to the newly added vertex. This feature makes the path more smooth.

Basically, we have the same parameter settings as the RRT algorithm. However, in the experiments, we chose different parameter settings to see how it will affect the result to what extent. More specifically, we change the probability that choose the actual goal state to attempt to go towards, and the maximum length of a motion to be added in the tree of motions. See section IV for more details.

In exchange for finding the optimal path, RRT* algorithm does not have the fastest search time. In fact, it will continue searching the optimal path until the user defined time is up. Therefore, in section IV, we did not include the search time of RRT* in the graph. In the next section, we will introduce RRTConnect algorithm, which usually perform faster search than both RRT and RRT* algorithm.

Algorithm 6 illustrates the general idea of RRT* algorithm for this particular problem, where V is the vertex and E is the tree edge.

G. RRTConnect Algorithm

In this section, we will introduce RRTConnect and discuss the parameters we specified for the algorithm.

This algorithm can think of a bidirectional version of RRT algorithm. Thus, it is expected to perform a faster time search. In fact, in section IV, we can indeed see that RRTConnect has a better search time overall. However, this algorithm does not guarantee optimal path search. Therefore, the resulting path length may not has huge difference when compared to the resulting path of the RRT.

For the parameter settings, we basically use the same parameter settings as the RRT algorithm, since we thought it would be best to compare the efficiency between those two algorithms.

Algorithm 6 RRT* Algorithm

```
1: $V \leftarrow \{x_s\}, E \leftarrow \emptyset$
2: for $i = 1 \ldots n$ do
3:     $x_{rand} \leftarrow SampleFree()$
4:     $x_{nearest} \leftarrow Nearest((V, E), x_{rand})$
5:     $x_{new} \leftarrow Steer(x_{nearest}, x_{rand})$
6:     if $CollisionFree(x_{nearest}, x_{new})$ then
7:         $x_{near} \leftarrow Near((V, E), x_{new}, min\{r^*, \epsilon\})$
8:         $V \leftarrow V \cup \{x_{new}\}$
9:         $c_{min} \leftarrow Cost(x_{nearest})$
10:                $+ Cost(Line(x_{nearest}, x_{new}))$
11:        for $x_{near} \in \mathbb{X}_{near}$ do
12:            if $CollisionFree(x_{nearest}, x_{new})$ then
13:                if $Cost(x_{near}) + Cost(Line(x_{near}, x_{new})) <$
                   $c_{min}$ then
14:                    $x_{min} \leftarrow x_{near}$
15:                    $c_{min} \leftarrow Cost(x_{near})$
16:                            $+ Cost(Line(x_{near}, x_{new}))$
17:                end if
18:            end if
19:        end for
20:        $E \leftarrow E \bigcup \{x_{(nearest}, x_{new})\}$
21:        for $x_{near} \in \mathbb{X}_{near}$ do
22:            if $CollisionFree(x_{new}, x_{near})$ then
23:                if $Cost(x_{new}) + Cost(Line(x_{new}, x_{near})) <$
                   $Cost(x_{near})$ then
24:                    $x_{parent} \leftarrow Parent(x_{near})$
25:                    $E \leftarrow (E \quad \{(x_{parent}, x_{near})\}) \cup$
                       $\{(x_{new}, x_{near})\}$
26:                end if
27:            end if
28:        end for
29:    end if
30: end for
31: return $G = (V, E)$
```

Algorithm 7 illustrates the general idea of RRTConnect algorithm for this particular problem, where V is the vertex and E is the tree edge.

IV. Results

In this section I will present the comparison between our search based algorithm and sampling based algorithms, as well as other interesting details.

A. Path visualization

As shown below, Figure 1 to 4 show the path visualization of different algorithms on each map. As you can see, the path of D* Lite is very intuitive. D* Lite attempts to look for a feasible path under a dynamic map, which means that it needs to "explore" the map and has a chance to detour. For example, the path in the map Flappy Bird clearly demonstrates the extensive exploration of the D* Lite, while the path in the

Algorithm 7 RRTConnect Algorithm

```
1: $V_a \leftarrow \{x_s\}, E_a \leftarrow \emptyset, V_b \leftarrow x_{tau}, E_b \leftarrow \emptyset$
2: for $i = 1 \dots n$ do
3:     $x_{rand} \leftarrow SampleFree()$
4:     if not $Extend((V_a, E_a), x_{rand}) = Trapped$ then
5:         if $Connect((V_b, E_b), x_{new}) = Reached$ then
6:             return $Path((V_a, E_a), (V_b, E_b))$
7:         end if
8:     end if
9:     $Swap((V_a, E_a), (V_b, E_b))$
10: end for
11: return $Failure$
```

map Window is more straightforward and precise. Whenever the map is a monotonic environment, it is hard for the agent to distinguish difference in the map and therefore takes more time to find a way. On the other hand, if the map is informative and consists of unique characteristics on each region, the results would be better than the average performance of RRT series.

From the results, we can tell the core difference between D* Lite and RRT series. RRT algorithms gives a faster, but less smoother path, while D* Lite demonstrates a more clear path. We can further examine the the observations through numbers. In the following section, we would discuss the difference of path length and consuming time using various algorithms.

B. Time and Path Length

Here we provide the computational time and solution path length for four maps (single cube, room, window, and flappy bird) over 30 samples.

From Figure 6 to 9 we can see that, for RRT* algorithm, the resulting solution path has much fewer path length comparing to other sampling based algorithm, the reason is that RRT* guarantee the optimality of the solution path while others does not. Moreover, for D* algorithm, the path length is always the same since it is search based algorithm. We also tried the different parameter setting to see the resulting difference in RRT*, Figure 10 shows that shorter length of a motion will result in longer optimal path length.

For search time, notice that we did not include the search time for RRT* since RRT* keep exploring optimal path until the user defined time is reached. Additionally, we can see that RRTConnect perform better than RRT in terms of search time. In Figure 5, we showed the search time of D* algorithm for 4 different maps, we put it in a different plot since the time scale of D* is different than those of sampling based algorithms.

C. Conclusion

From the results above, we can see that our D* Lite and all sampling based algorithm works well. However, we think that the D* solution can be improved in some ways. Below are some possible improvements:

- In our implementation, the function updateNeighbor() costs the most in the algorithm since that it needs to scan and update all neighbors of a given node and updating each neighbor requires extra iterations to find the minimal cost. In this project, we use for loop to realize the function. However, it would be much faster if we could use parallel computation instead.
- In our default settings, all edge cost is set to be one initially, and the cost accumulates through the process and turns to infinity while encountering an obstacle. However, if we could assign the cost with more conditions. The algorithm might works better since that the change of cost gives more information than before. However, it is a challenging request because we are dealing with unknown, dynamic map and therefore the information we can collect in this scenario is limited.
- The size of observed region from the agent in each iteration affects how the path is derived. The larger the region is, the fewer the agent would detour. However, larger region the agent observes, more time and computation it takes. As a result, it is recommended to decide a suitable size before executing mass computation on a large map.

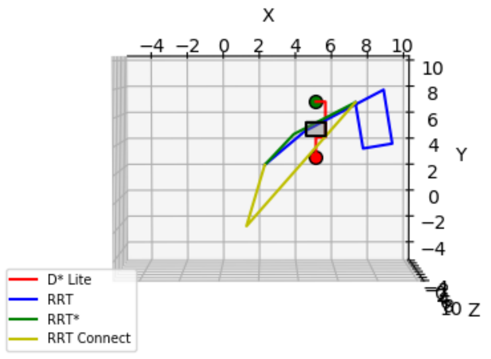


Fig. 1: Four resulting paths for Single Cube

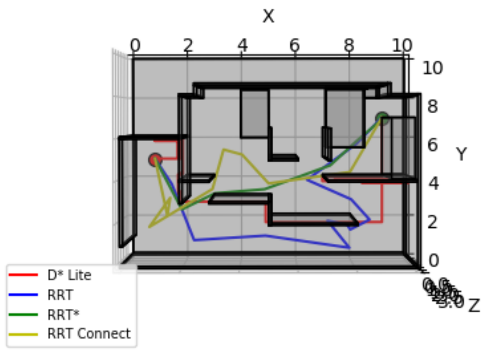


Fig. 2: Four resulting paths for Room

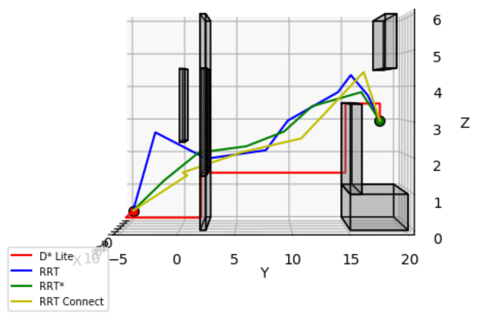


Fig. 3: Four resulting paths for Window

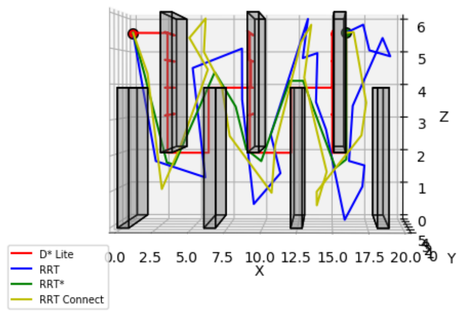


Fig. 4: Four resulting paths for Flappy Bird

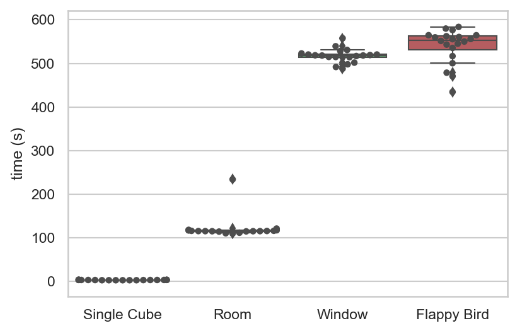
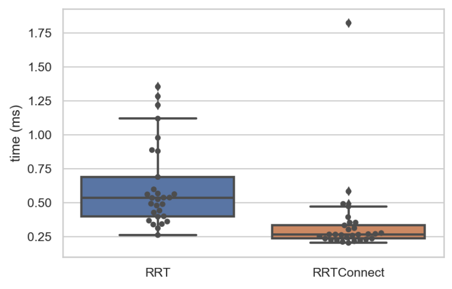
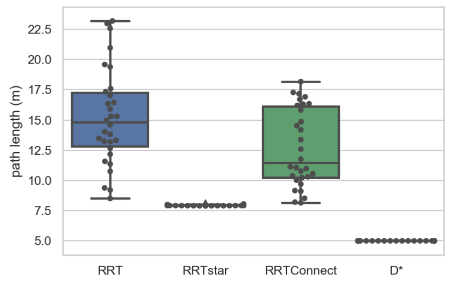


Fig. 5: Search time of D* for four different maps

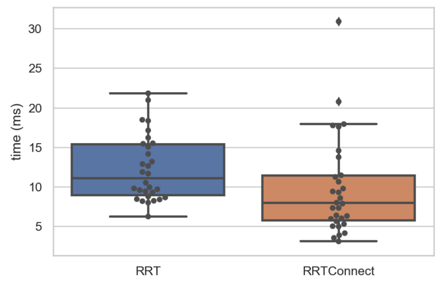


(a) Search Time

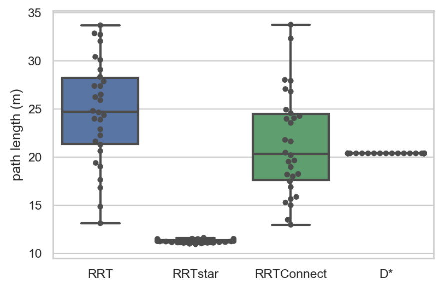


(b) Path Length

Fig. 6: Single Cube

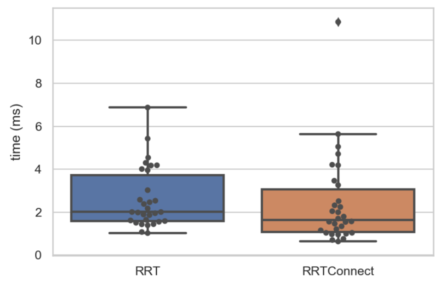


(a) Search Time

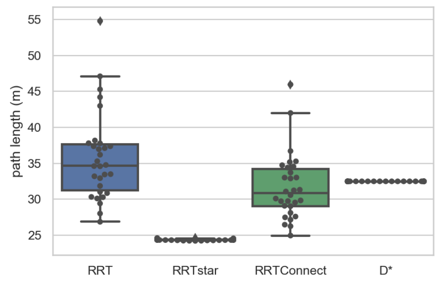


(b) Path Length

Fig. 7: Room

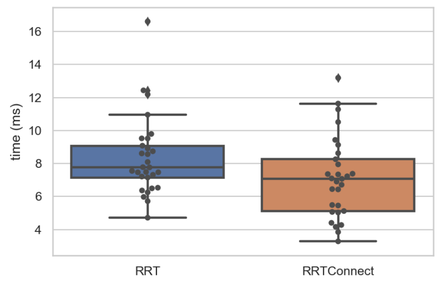


(a) Search Time

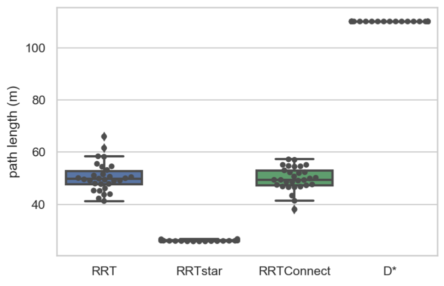


(b) Path Length

Fig. 8: Window



(a) Search Time



(b) Path Length

Fig. 9: Flappy Bird

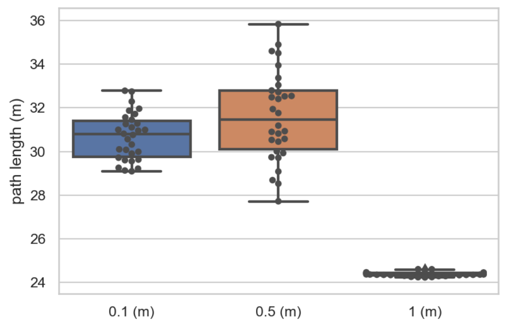


Fig. 10: Optimal Path Length for different range settings

References

[1] Joe Leavitt, Ben Ayton, Jessica Noss, Erlend Harbitz, Jake Barnwell, and Sam Pinto (2016). *Incremental Path Planning*. Massachusetts Institute of Technology

[2] Sven Koenig and Maxim Likhachev(2002). *D* Lite*. American Association for Artificial IntelliGence